Double Backpropagation
Increasing Generalization Performance

Harris Drucker
AT&T Bell Laboratories and Monmouth College
mailing address: Monmouth College, West Long Branch, NJ 07764

Yann Le Cun
AT&T Bell Laboratories
Room 3G-332, Crawford Corners Road, Holmdel, NJ 07733

ABSTRACT

One test of a new training algorithm is how well the algorithm generalizes from the training data to the test data. A new training algorithm termed double backpropagation improves generalization by simultaneously minimizing the normal energy term found in backpropagation and an additional energy term that is related to the sum of the squares of the input derivatives (gradients). In normal backpropagation training, minimizing the energy function tends to push the input gradient to zero. However this is not always possible. Double backpropagation explicitly pushes the input gradients to zero making the minimum broader and increases the generalization on the test data. We show the improvement over normal backpropagation on four candidate architectures and a training set of 320 handwritten numbers and a test set of size 180.

Introduction

Backpropagation has been discussed extensively [1] and so we will not recreate the general algorithm here. Rather, let us examine a particular architecture (Figure 1 - below the dashed line) with three input neurons, two hidden-layer neurons and two output neurons. The layers are fully connected (to the layer immediately above) but not all weights are shown (nor is the bias). We use letters (rather than the typical weight notation with superscripts and subscripts) to denote the weights. The input layer has linear neurons while the other layers have the nonlinear transfer function. Therefore $a_j^{(r)}$, represents the summed input of neuron number j at layer r and $x_j^{(r)} = f\left[a_j^{(r)}\right]$ where f is the hyperbolic tangent.

In normal backpropagation we first propagate the state of the input forward through the layers to the output. We then form an energy (or error) term which in this case is:

$$E_f = \frac{1}{2}\left[d_1 - x_1^{(2)}\right]^2 + \frac{1}{2}\left[d_2 - x_2^{(2)}\right]^2 \tag{1}$$

where d is the desired output. We call this the forward energy function because it is obtained by propagating the states forward through the network. Now, let us look at some of the terms obtained by backpropagating through the network. First, the derivative of the forward energy function with respect to the input of the first neuron of the output layer:

$$-\frac{\partial E_f}{\partial a_1^{(2)}} = \left[d_1 - x_1^{(2)}\right] f'\left[a_1^{(2)}\right]. \tag{2}$$

where f′ is the derivative. Next, the derivatives of the forward energy function with respect to the input of both neurons at the hidden layer:

$$\frac{\partial E_f}{\partial a_1^{(1)}} = \left[A\frac{\partial E_f}{\partial a_1^{(2)}} + C\frac{\partial E_f}{\partial a_2^{(2)}}\right] f'\left[a_1^{(1)}\right] \qquad \frac{\partial E_f}{\partial a_2^{(1)}} = \left[B\frac{\partial E_f}{\partial a_1^{(2)}} + D\frac{\partial E_f}{\partial a_2^{(2)}}\right] f'\left[a_2^{(1)}\right].$$

The change in weight A is now due to the term $-\frac{\partial E_f}{\partial a_1^{(2)}}$ multiplied by both $x_1^{(1)}$ and the learning constant. Similarly we can update the other weights. In normal backpropagation, these are all the gradients needed.

0-7803-0164-1/91/0000-0145$01.00©1991 IEEE

However, we can proceed one more step and calculate the derivative of the forward energy function with respect to the input: $\frac{\partial E_f}{\partial i_1}$. Therefore the weights A through F are all involved in the calculation of the input gradient. If, after training, all the gradients at the output layer are zero, then the input gradients will be zero. Of course, this is invariably never true. We would however like to force the input gradients to be zero even if the preceding gradients are nonzero. This would tend to make the output less sensitive to amplitude variations of the input and give a smoother approximation.

Double Backpropagation

Normal backpropagation can be viewed as propagating states through a network, only backwards. Therefore we can take the output of the network and construct an appended network (the top part of Figure 1) that calculates the input gradient. The appended network uses linear neurons:

$$y_j^{(r)} = k_j^{(r)} b_j^{(r)}$$

$$k_j^{(r)} = \begin{cases} f'(a_j^{(r)}) & r > 0 \\ 1 & r = 0 \end{cases}$$

where k is the multiplicative constant whose value is related to the derivative of the input state of a neuron in the lower network.

Although not all weights are shown consider the input state of the first neuron of the appended layer:

$$b_1^{(2)} = \left[x_1^{(2)} - d_1 \right]$$

$$y_1^{(2)} = k_1^{(2)} b_1^{(2)}$$

$$= f' \left[a_1^{(2)} \right] \left[x_1^{(2)} - d_1 \right]$$

(which is just the first term in using the backpropagation algorithm (equation (2)). Proceeding in this fashion, we see that the output of the appended network is just the input gradient of the forward energy function. Note that the bias is not involved in the calculation of the input gradient. Therefore, three steps are involved in calculating the input gradient: (1) forward propagate through the lower network (2) copy the derivatives of the input states from the lower network to the appended network as the multiplicative constants of the linear neurons and (3) forward propagate through the appended network. Incidentally, once training is over, we remove the appended network.

Now we can form another energy function, termed the backward energy function, so named because it could be obtained by backpropagating through the lower network:

$$E_b = \left[t_1 - y_1^{(0)} \right]^2 + \frac{1}{2} \left[t_2 - y_2^{(0)} \right]^2 + \frac{1}{2} \left[t_3 - y_3^{(0)} \right]^2$$

$$= \frac{1}{2} \left[t_1 - \frac{\partial Ef}{\partial i_1} \right]^2 + \frac{1}{2} \left[t_2 - \frac{\partial Ef}{\partial i_2} \right]^2 + \frac{1}{2} \left[t_3 - \frac{\partial Ef}{\partial i_3} \right]^2 \qquad (3)$$

where the t's are the target derivatives. We will now minimize both the backward energy function (3) and the forward energy function (1). The general idea is to backpropagate through the lower network to minimize the forward energy function and do another backpropagation starting at the top of the appended network to minimize the backward energy function (hence the description of the training algorithm as double backpropagation).

Backpropagation through the upper network has some subtleties because the weights are shared between the upper network and the lower network. First let us find the derivative of the backward energy function

with respect to the state of the first neuron in the top layer recalling both that the neuron is linear and that for the top layer, the multiplicative constant is 1.

$$\frac{\partial Eb}{\partial b_1^{(0)}} = -\left(t_1 - y_1^{(0)}\right).$$

Now the approximate gradient with respect to the weight F:

$$\frac{\partial Eb}{\partial F} = \frac{\partial Eb}{\partial b_1^{(0)}} \frac{\partial b_1^{(0)}}{\partial F} + \frac{\partial Eb}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial F} = \frac{\partial Eb}{\partial b_1^{(0)}} y_2^{(1)} + \frac{\partial Eb}{\partial a_2^{(1)}} i_1 .$$

The first term of the sum is found in normal backpropagation, the second is not. However, we will get the equivalent result by backpropagating through the whole network. Therefore, the algorithm proceeds as follows:

1. Present the input pattern and propagate it to the output (the top of the lower network).
2. Backpropagate the derivative of the forward energy function through the lower network. Compute the change in weights but do not change the weights yet.
3. Copy the appropriate derivatives from the lower network to the appended network.
4. Propagate forward through the upper network
5. Now backpropagate the backward energy function through the entire network using the target derivatives, calculating the weights changes but do not change the weights yet.
6. Finally, change the weights using the weight changes calculated in steps 2 and 5.

Theoretically any target derivative can be picked but we chose the target derivatives to be zero for the following experiments.

Experimental Results

We used a database consisting of 48 sets of the ten numbers written using a mouse, the first 32 sets used for training and the remainder for testing. The input is on a 16 by 16 pixel array with -1's for the background and +1 for the foreground. We used four architectures which have strong local constraints that have been shown to give good generalization [2]:

1. local-net: A locally connected architecture with two hidden layers. The input layer is 16x16 and the first hidden layer is of size 8x8 with each neuron on the hidden layer taking its input from a 3x3 square on the input layer. For units in the hidden layer that are one unit apart, their receptive fields (in the input layer) are two pixels apart. Thus the receptive fields in the hidden layer overlap by one column and one row. The second hidden layer is of size 4x4 with a receptive field of size 5x5, overlapping again by one column and one row. The second hidden layer is fully connected to the output.
2. local2-net: A network with two hidden layers and weight sharing. There are two hidden layers with the first hidden layer consisting of two 8x8 feature maps, each feature map examining a 3x3 neighborhood on the input layer. All units in a feature map share the same weights. The second hidden layer is again of size 4x4 with receptive fields of size 5x5 and no weight sharing. The output of the second hidden layer is fully connected to the output.
3. local21-net: Same as local2-net except that the weights in going from the first to second hidden layer are shared.
4. local4-net: Two hidden layers, the first of which consists of four 8x8 feature maps and the second four 4x4 feature maps. All the units in a feature map share the same weights, the receptive fields being of size 3x3 in going from the first to second hidden layer and of size 5x5 going from the first to second hidden layer.

Because using a double backpropagation algorithm is computationally expensive, we are interested in determining whether we can train by running normal backpropagation and then follow with a limited number of runs using double backpropagation. The results presented below are for each of the architectures above and ten runs per architecture. A single run consisted of the following:

(1) initializing the architecture to some set of random weights.

(2) training on the 320 characters (60 iterations) using backpropagation and then testing on the 180 characters.

(3) continued training on the 320 characters using double backpropagation (another 25 iterations) and then tested.

(4) reinitializing to the original set of weights and trained using double backpropagation only (75 iterations).

Another set of variables is the choice of learning constants. Recall that the output layer is defined as the output of the normal architecture, not the appended network and that the double backpropagation algorithm consists of backpropagating once from the output (using the desired state) towards the input and once from the top of of the appended network (using the target derivatives). The learning constants in each of these backward passes is not necessarily the same. We tried two combinations of learning constants, the first combination such that the constants were the same in both backward passes and the second having the learning constant in the backwards pass from the top of the appended network twice that of the pass backwards from the output layer. This latter combination forces the network faster towards the target derivatives than the desired states and invariably gives better results and so just these results are given in the two tables.

In Table I the first six columns give the average, minimum, and maximum generalization performance for the ten runs for backpropagation or backpropagation followed by double backpropagation training. The next set of three columns relates to the improvement of backpropagation followed by double backpropagation over backpropagation alone. For each of the ten runs, there is an improvement found by subtracting the score obtained using backpropagation from the score using double backpropagation. Of these ten differences, there is a maximum (Δ max) difference and minimum difference (Δ min) and an average of the differences (Δ avg). Thus for the local-net architecture, on the average the improvement in recognition scores is 1.7% while the smallest and largest improvements were -1.3 and 4.4% respectively (where a negative sign indicates a decrease in performance). Finally the last column indicates in how many cases the backprop followed by double backprop gives a result better than backprop alone. Although the results on the training sets are not shown it is interesting that training with backpropagation alone always give 100% recognition on the training set, Using double backpropagation, the error rate is not always zero on the training set even though generally double backpropagation gives better results.

Table II compares backpropagation against double backpropagation. The columns are the same as Table I except there is an extra column which indicates how many cases in which double backpropagation is better than backprop followed by double backpropagation. How much better can be obtained from comparing column seven of both tables.

Conclusions

These experiments show that double backpropagation either alone or following normal training using backpropagation can lead to improved scores. A possible disadvantage is longer training times especially when the network or training data is large. Normal training using backpropagation followed by double backpropagation is an alternative. The results are especially impressive since at the general high levels of performance using backpropagation alone, it is difficult to achieve increases for the same architecture.

REFERENCES

[1] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning internal representation by error propagation, in Parallel Distributed Processing," Vol. 1 Ch. 8, Rumalhardt and McClelland, eds. MIT Press.

[2] Y. LeCun, Generalization and Network Design Strategies. In Pfeifer, Schreteer, Fogelman, and Steels, eds., Connectionism in Perspective, Zurich, Switzerland. Elsevier.

	BACKPROP			BACKPROP-THEN-DOUBLE BACKPROP						
	AVG	MIN	MAX	AVG	MIN	MAX	Δ AVG	Δ MIN	Δ MAX	N_1
LOCAL	89.5	85.6	92.5	91.2	86.8	94.4	1.7	-1.3	4.4	9/10
LOCAL 2	94.3	90.6	96.9	96.5	94.4	98.1	2.2	0.6	4.3	10/10
LOCAL 21	91.0	86.9	95.0	93.8	91.8	95.6	2.8	0.6	5.0	10/10
LOCAL 4	94.1	91.9	96.8	97.1	95.0	99.4	3.0	0.7	5.4	10/10

TABLE I: Backpropagation followed by double backpropagation versus backpropagation only. Scores are in per cent correct. N_1 is number of cases when the former is greater than the latter.

	BACKPROP			DOUBLE BACKPROPAGATION							
	AVG	MIN	MAX	AVG	MIN	MAX	Δ AVG	Δ MIN	Δ MAX	N_1	N_2
LOCAL	89.5	85.6	92.5	94.9	91.3	97.5	5.42	1.3	9.4	10/10	10/10
LOCAL 2	94.3	90.6	96.9	96.1	94.4	97.5	1.9	-1.3	5.0	7/10	4/10
LOCAL 21	91.0	86.9	95.0	93.0	90.6	93.8	2.0	-2.5	6.9	9/10	4/10
LOCAL 4	94.1	91.9	96.8	95.5	94.0	96.7	1.4	-1.5	3.5	9/10	1/10

TABLE II: Double backpropagation vs backpropagation N_1 is number of runs where double backpropagation is better than backpropagation. N_2 is number of runs where double backpropagation is better than backpropagation followed by double backpropagation.

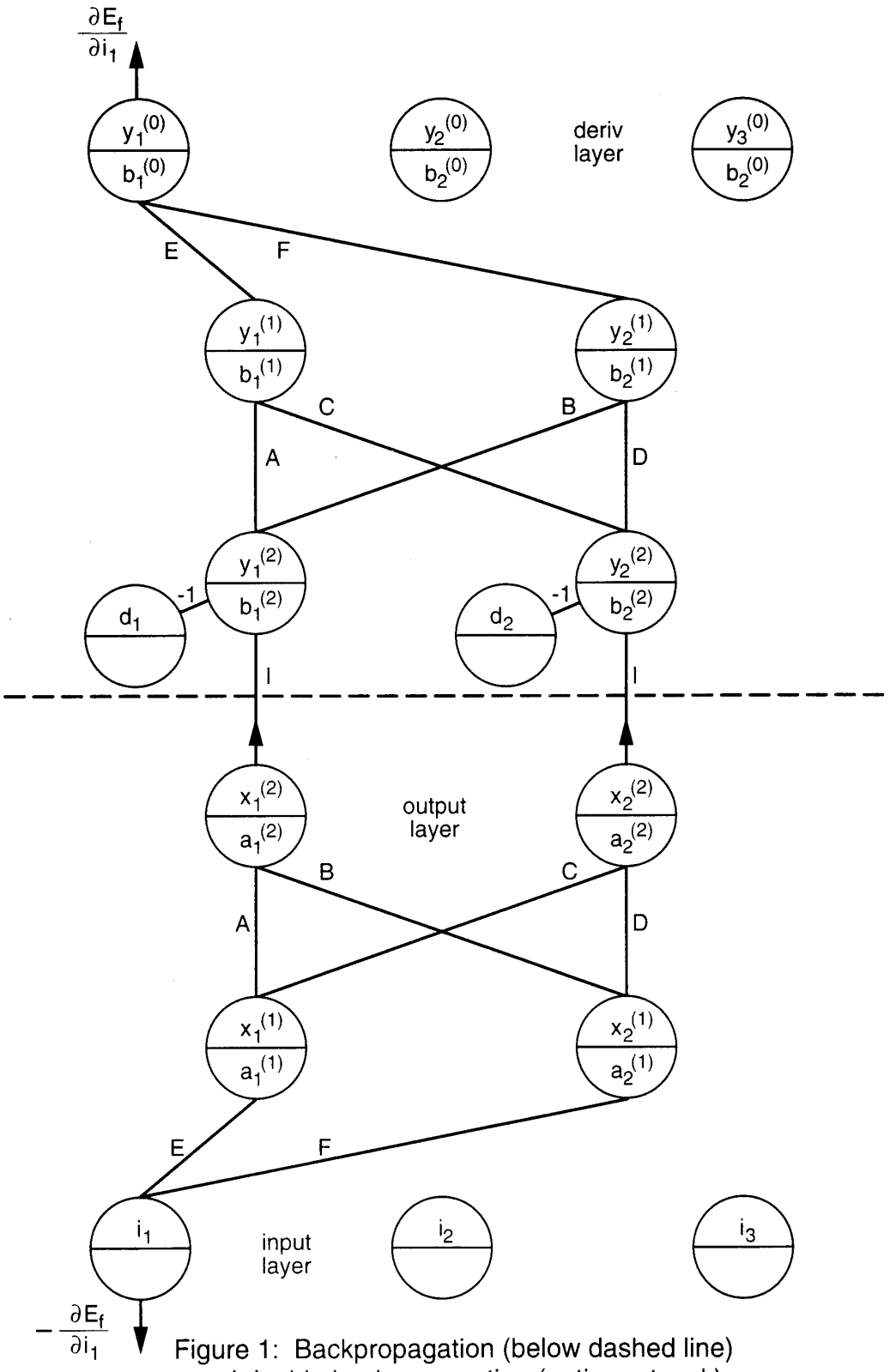

Figure 1: Backpropagation (below dashed line) and double backpropagation (entire network)

20095.03